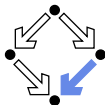
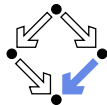


# Specifying in the Large

Wolfgang Schreiner  
Wolfgang.Schreiner@risc.jku.at

Research Institute for Symbolic Computation (RISC)  
Johannes Kepler University, Linz, Austria  
<http://www.risc.jku.at>





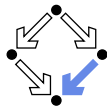
---

# 1. A Specification Language

## 2. Modularization

## 3. Parameterization

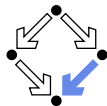
## 4. Further Topics



# A Specification Language

A language for building “large” specifications from “small” ones.

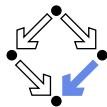
- **Abstract Syntax:** set  $SL$  of specifications  $sp$  with signatures  $\mathcal{S}(sp)$ .
  - **Atomic:** If  $sp$  is “atomic” (a specification as previously defined), then
$$sp \in SL$$
with  $\mathcal{S}(sp)$  as previously defined.
  - **Union:** If  $sp_1 \in SL$  and  $sp_2 \in SL$ , then
$$(sp_1 + sp_2) \in SL$$
with  $\mathcal{S}(sp_1 + sp_2) = \mathcal{S}(sp_1) \cup \mathcal{S}(sp_2)$ .
  - **Renaming:** If  $sp \in SL$  and  $\mu : \mathcal{S}(sp) \rightarrow \Sigma'$  is a renaming, then
$$(\text{rename } sp \text{ by } \mu) \in SL$$
with  $\mathcal{S}(\text{rename } sp \text{ by } \mu) = \mu(\mathcal{S}(sp))$ .
  - **Forgetting:** If  $sp \in SL$ ,  $S$  is a set of sorts and  $\Omega$  is a set of operations such that  $(S, \Omega) \subseteq \mathcal{S}(sp)$  and  $\mathcal{S}(sp) \setminus (S, \Omega)$  is a signature, then
$$(sp \text{ forget } (S, \Omega)) \in SL$$
with  $\mathcal{S}(sp \text{ forget } (S, \Omega)) = \mathcal{S}(sp) \setminus (S, \Omega)$ .
  - ...



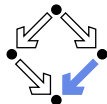
# A Specification Language (Contd)

- **Abstract Syntax:** set  $SL$  of specifications  $sp$  with signatures  $\mathcal{S}(sp)$ .
  - ...
  - **Extension:** If  $sp \in SL$ ,  $S$  is a set of sorts and  $\Omega$  is a set of operations such that  $\mathcal{S}(sp) \cup (S, \Omega)$  is a signature, then
$$(sp \text{ extend } (S, \Omega)) \in SL$$
with  $\mathcal{S}(sp \text{ extend } (S, \Omega)) = \mathcal{S}(sp) \cup (S, \Omega)$ .
  - **Modelling:** if  $sp \in SL$  and  $\Phi \subseteq L(\mathcal{S}(sp))$  for some logic  $L$ , then
$$(sp \text{ model } \Phi) \in SL$$
with  $\mathcal{S}(sp \text{ model } \Phi) = \mathcal{S}(sp)$ .
  - **Restricting:** if  $sp \in SL$  with  $\mathcal{S}(sp) = (S, \Omega)$ , if  $S_c \subseteq S$  is a set of sorts and if  $\Omega_c \subseteq \Omega$  is a set of operations with target sorts in  $S_c$ , then
$$(sp \text{ generated in } S_c \text{ by } \Omega_c) \in SL \text{ and}$$
$$(sp \text{ freely generated in } S_c \text{ by } \Omega_c) \in SL$$
with  $\mathcal{S}(sp \text{ generated in } S_c \text{ by } \Omega_c) = \mathcal{S}(sp)$ and  $\mathcal{S}(sp \text{ freely generated in } S_c \text{ by } \Omega_c) = \mathcal{S}(sp)$ .

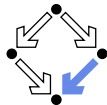
$\mathcal{S}(sp)$  is a signature for any specification  $sp \in SL$ .



- $(S, \Omega)$  :  
    **sorts** *sorts*  
    **opns** *operations*
- $\mu : \Sigma \rightarrow \Sigma'$   
    **sorts**  $s_1, \dots, s_k$  **opns**  $\omega_1, \dots, \omega_l$  **as**  
    **sorts**  $s'_1, \dots, s'_k$  **opns**  $\omega'_1, \dots, \omega'_l$
- Example:  $\mathcal{S}(sp) = (\{s, t\}, \{m : s \times t \rightarrow s, n : t \times s \rightarrow t, n : \rightarrow s\})$ .  
    (**rename** *sp*  
    **by sorts**  $s$  **opns**  $n : t \times s \rightarrow t$   
    **as sorts**  $u$  **opns**  $q : t \times u \rightarrow t$ )  
    means (**rename** *sp* **by**  $\mu$ ) with  $\mu : \Sigma \rightarrow \Sigma'$  defined as  
     $\Sigma = \mathcal{S}(sp), \Sigma' = \mu(\Sigma)$   
     $\mu(s) = u, \mu(t) = t$   
     $\mu(m : s \times t \rightarrow s) = (m : u \times t \rightarrow u)$   
     $\mu(n : t \times s \rightarrow t) = (q : t \times u \rightarrow t)$   
     $\mu(n : \rightarrow s) = (n : \rightarrow u)$



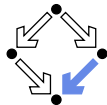
- **Semantics:**  $\mathcal{M}(sp)$  is inductively defined:
  - $\mathcal{M}(sp)$  of an atomic specification  $sp$  is as previously defined;
  - $\mathcal{M}(sp_1 + sp_2) = \{A \in Alg(\mathcal{S}(sp_1 + sp_2)) \mid (A|\mathcal{S}(sp_1)) \in \mathcal{M}(sp_1), (A|\mathcal{S}(sp_2)) \in \mathcal{M}(sp_2)\};$   
 $A|\Sigma \dots \Sigma$ -reduct of  $A$ 
    - Hide sorts and operations that do not occur in signature  $\Sigma$ .
  - $\mathcal{M}(\text{rename } sp \text{ by } \mu) = \{A \in Alg(\mu(\mathcal{S}(sp))) \mid (A|\mu) \in \mathcal{M}(sp)\};$   
 $A|\mu \dots \mu$ -reduct of  $A$ 
    - Rename sorts and operations as indicated by renaming  $\mu$ .
  - $\mathcal{M}(sp \text{ forget } (S, \Omega)) = \mathcal{M}(sp) \mid (\mathcal{S}(sp) \setminus (S, \Omega));$
  - $\mathcal{M}(\text{extend } sp \text{ by } (S, \Omega)) =$   
 $\{A \in Alg(\mathcal{S}(sp) \cup (S, \Omega)) \mid (A|\mathcal{S}(sp)) \in \mathcal{M}(sp)\};$
  - $\mathcal{M}(sp \text{ model } \Phi) = \mathcal{M}(sp) \cap Mod_{\mathcal{S}(sp)}(\Phi);$
  - $\mathcal{M}(sp \text{ generated in } S_c \text{ by } \Omega_c) =$   
 $\{A \in \mathcal{M}(sp) \mid A \text{ is generated in } S_c \text{ by } \Omega_c\};$
  - $\mathcal{M}(sp \text{ freely generated in } S_c \text{ by } \Omega_c) =$   
 $\{A \in \mathcal{M}(sp) \mid A \text{ is freely generated in } S_c \text{ by } \Omega_c\}.$



- **Operator** + builds the “union” of two specifications  $sp_1$  and  $sp_2$ .
  - If  $sp_1$  and  $sp_2$  have common sorts/operations, only those algebras of  $\mathcal{M}(sp_1)$  and  $\mathcal{M}(sp_2)$  contribute to this union that have the same interpretation of the common parts.
- **rename** may be used to avoid “name clashes”.
  - If two specifications have the same sort/operator with different meaning, rename this entity in one of them before constructing the union of both specifications.
- **forget** hides sorts and operations.
  - For auxiliary entities that are not part of the “public” specification interface.
- **extend** introduces new sorts and operations.
  - Loose semantics of new entities.
- **model** and **(freely) generated by** filter out unintended algebras.

# Properties

---



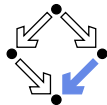
Take specification  $sp \in SL$ .

- Every algebra in  $\mathcal{M}(sp)$  has signature  $\mathcal{S}(sp)$ .
- $\mathcal{M}(sp)$  is an abstract datatype.

The semantics of the specification language is “as expected”.



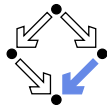
# Example



```
(extend (  
  (loose spec  
    sorts freely generated bool  
    opns constr True :→ bool, False :→ bool  
  endspec +  
  loose spec  
    sorts nat  
    opns 0 :→ nat, Succ : nat → nat  
  endspec)  
  freely generated  
  in sorts nat  
  by opns 0 :→ nat, Succ : nat → nat)  
by opns _ ≤ _ : nat × nat → bool)  
model vars m, n : nat  
axioms  
   $0 \leq n = \text{True}$   
   $\text{Succ}(m) \leq 0 = \text{False}$   
   $\text{Succ}(m) \leq \text{Succ}(n) = m \leq n$ 
```

A (still rather clumsy) specification of the “classical” algebra.

# A Specification Language with Environments

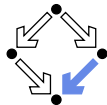


Introduce an **environment**  $e$  that maps names to specifications.

- **Abstract syntax:** set  $SL(e)$  of specs  $sp$  with signatures  $\mathcal{S}(e, sp)$ .
  - If  $n$  is a name such that  $e(n)$  is defined, then
$$n \in SL(e)$$
with  $\mathcal{S}(e, n) = \mathcal{S}(e, e(n))$ .
  - ... (as before)
    - Using  $SL(e)$  and  $\mathcal{S}(e, sp)$  rather than  $SL$  and  $\mathcal{S}(sp)$ .
- **Semantics:**  $\mathcal{M}(e, sp)$  is inductively defined:
  - $\mathcal{M}(e, n) = \mathcal{M}(e, e(n))$
  - ... (as before)
    - Using  $\mathcal{M}(e, sp)$  and  $\mathcal{S}(e, sp)$  rather than  $\mathcal{M}(sp)$  and  $\mathcal{S}(sp)$ .

**Specifications can be named.**

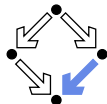
# Concrete Syntax



- **Environment:** defined by a declaration (sequence).
  - $\epsilon$ : the empty declaration sequence.
    - Denoting the environment that does not contain any mapping.
  - $n$  **is**  $sp$ : a sequence with a single declaration.
    - Denoting the environment that only maps  $n$  to  $sp$ .
  - $d; n$  **is**  $sp$ : declaration sequence  $d$  followed by a declaration.
    - Denoting the environment that maps  $n$  to  $sp$  and every other name to the same specification as the environment denoted by  $d$  does.
- **Specification:**  $d; sp$ 
  - Declaration (sequence)  $d$  denoting an environment  $e$ .
  - $sp \in SL(e)$ .
  - Special case:  $\epsilon; sp$  is simply written as  $sp$ .

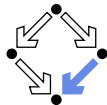
Specifications are defined in the context of declarations.

# Example



```
BOOL is
  loose spec
    sorts freely generated bool
    opns constr True :→ bool, False :→ bool
  endspec;
NAT is
  loose spec
    sorts nat
    opns 0 :→ nat, Succ : nat → nat
  endspec;
BOOLNAT is BOOL + NAT
  freely generated
  in sorts nat
  by opns 0 :→ nat, Succ : nat → nat;
extend BOOLNAT by opns  $\leq$  : nat × nat → bool
model vars m, n : nat
  axioms
    0 ≤ n = True
    Succ(m) ≤ 0 = False
    Succ(m) ≤ Succ(n) = m ≤ n
```

A structured specification of the “classical” algebra.



---

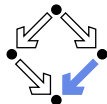
## 1. A Specification Language

## 2. Modularization

## 3. Parameterization

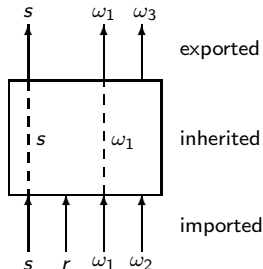
## 4. Further Topics

# Module Signatures

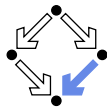


A module is an entity with a well-defined interface to its environment.

- **Module signature:** pair  $(\Sigma_i, \Sigma_e)$ .
  - **Import signature**  $\Sigma_i$ .
    - A sort/operation from  $\Sigma_i$  is called **imported**.
  - **Export signature**  $\Sigma_e$ .
    - A sort/operation from  $\Sigma_e$  is called **exported**.
  - A sort/operation from  $\Sigma_i \cap \Sigma_e$  is called **inherited**.
- **Example:**  $\Sigma_i = (\{r, s\}, \{\omega_1, \omega_2\})$ ,  $\Sigma_e = (\{s\}, \{\omega_1, \omega_3\})$ .

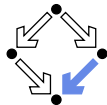


# Modularized Abstract Datatypes



Take module signature  $(\Sigma_i, \Sigma_e)$ .

- A  $(\Sigma_i, \Sigma_e)$ -module (also called a “modularized abstract datatype”)  $M : Alg(\Sigma_i) \rightarrow \mathbb{P}(Alg(\Sigma_e))$ 
  - is a mapping from  $\Sigma_i$ -algebras to classes of  $\Sigma_e$ -algebras such that
  - for every  $A \in Alg(\Sigma_i)$ ,  $M(A) \subseteq Alg(\Sigma_e)$  is an abstract datatype.
- A  $(\Sigma_i, \Sigma_e)$ -module  $M$  is **persistent for an algebra**  $A \in Alg(\Sigma_i)$ , if  $\forall B \in M(A) : (A|_{\Sigma_i \cap \Sigma_e} \simeq (B|_{\Sigma_i \cap \Sigma_e}))$ .
  - Inherited sorts/operations have the same meaning in  $A$  and in  $M(A)$ .
- A  $(\Sigma_i, \Sigma_e)$ -module  $M$  is **consistent for an algebra**  $A \in Alg(\Sigma_i)$ , if  $M(A) \neq \emptyset$ .
  - The mapping  $M$  is “effective”.
- A  $(\Sigma_i, \Sigma_e)$ -module  $M$  is **monomorphic for an algebra**  $A \in Alg(\Sigma_i)$ , if  $M(A)$  is monomorphic.
- $M$  is **persistent/consistent/monomorphic**, if
  - it is consistent/persistent/monomorphic for every  $A \in Alg(\Sigma_i)$ .



# Loose Module Specifications

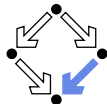
Take logic  $L$ .

- **Abstract syntax:** a loose module specification is a pair  $sp = ((\Sigma_i, \Sigma_e), \Phi)$  consisting of
  - a module signature  $(\Sigma_i, \Sigma_e)$  with  $\Sigma_i \subseteq \Sigma_e$ , and
  - a set of formulas  $\Phi \subseteq L(\Sigma_e)$ .
    - Entities of  $\Sigma_i$  are specified “elsewhere”.
- **Semantics:** the meaning of a loose module specification  $sp = ((\Sigma_i, \Sigma_e), \Phi)$  is the  $(\Sigma_i, \Sigma_e)$ -module defined as
$$\mathcal{M}(sp)(A) = \{B \in Alg(\Sigma_e) \mid B \models \Phi \wedge B|_{\Sigma_i} \simeq A\}$$
for every  $A \in Alg(\Sigma_i)$ .

A loose module specification defines a persistent (but not necessarily consistent) module.



# Concrete Syntax



$\Sigma_i = (\{bool, el\}, \{True, False\}), \Sigma_e = \Sigma_i \cup (\{list\}, \{[ ], Add, .\})$ .

**loose mspec**

**sorts** **import** *bool*, **import** *el*, *list*

**opns**

**import** *True*  $\rightarrow bool$

**import** *False*  $\rightarrow bool$

$[ ]$   $\rightarrow list$

*Add*  $: el \times list \rightarrow list$

$..$   $: list \times list \rightarrow list$

**vars** *l*, *m*  $: list$ , *e*  $: el$

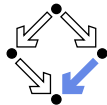
**axioms**

$[ ].l = l$

*Add*(*e*, *l*).*m* = *Add*(*e*, *l.m*)

**endspec**

Elements of the import signature are prefixed by the keyword **import**.



# A Module Specification Language

- **Abstract syntax:** set  $MSL$  of specs  $sp$  with signatures  $\mathcal{S}(sp)$ :
  - If  $sp$  is a loose module specification, then
$$sp \in MSL$$
with  $\mathcal{S}(sp)$  as previously defined;
  - If  $sp_1, sp_2 \in MSL$  with  $\mathcal{S}(sp_1) = (\Sigma_{1i}, \Sigma_{1e})$  and  $\mathcal{S}(sp_2) = (\Sigma_{2i}, \Sigma_{2e})$ 
    - and each sort and operation of  $\Sigma_{1e} \cap \Sigma_{2i}$  is inherited in  $\mathcal{S}(sp_1)$ ,
    - and each sort and operation of  $\Sigma_{2e} \cap \Sigma_{1i}$  is inherited in  $\mathcal{S}(sp_2)$ ,

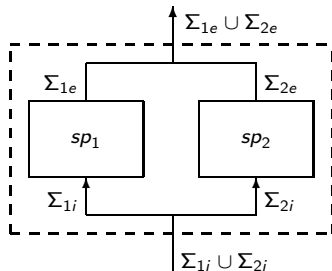
(no sort/operation introduced by one specification is imported by the other one)

then

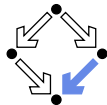
$$(sp_1 + sp_2) \in MSL$$

with  $\mathcal{S}(sp_1 + sp_2) =$   
 $(\Sigma_{1i} \cup \Sigma_{2i}, \Sigma_{1e} \cup \Sigma_{2e});$

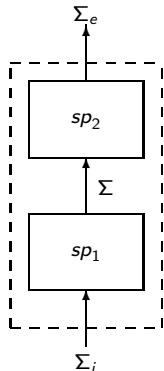
■ ...



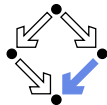
# A Module Specification Language (Contd)



- **Abstract syntax:** set  $MSL$  of specs  $sp$  with signatures  $\mathcal{S}(sp)$ :
  - ...
  - If  $sp_1, sp_2 \in MSL$  with  $\mathcal{S}(sp_1) = (\Sigma_i, \Sigma)$  and  $\mathcal{S}(sp_2) = (\Sigma, \Sigma_e)$ , then  $(sp_2 \circ sp_1) \in MSL$  with  $\mathcal{S}(sp_2 \circ sp_1) = (\Sigma_i, \Sigma_e)$ .
  - If  $sp \in MSL$  with  $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$  and  $\mu : \Sigma_e \rightarrow \Sigma'$  is a renaming with  $\mu(a) \notin \Sigma_i$  for each sort/operation  $a$  with  $\mu(a) \neq a$ , then  $(\text{rename } sp \text{ by } \mu) \in MSL$  with  $\mathcal{S}(\text{rename } sp \text{ by } \mu) = (\Sigma_i, \mu(\Sigma_e))$ ;  
(no clash between imported sorts/operations and "new" exported sorts/operations)
  - The constructs **forget**, **extend**, **model**, and **(freely) generated** are defined similarly as before.



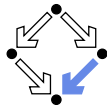
The language  $SL$  can be considered as a sublanguage of  $MSL$  where all module specifications have empty import signatures.



- **Semantics:**  $\mathcal{M}(sp)$  is inductively defined:
  - $\mathcal{M}(sp)$  of a loose module specification  $sp$  is as previously defined;
  - If  $\mathcal{S}(sp_1) = (\Sigma_{1i}, \Sigma_{1e})$  and  $\mathcal{S}(sp_2) = (\Sigma_{2i}, \Sigma_{2e})$ , then
$$\mathcal{M}(sp_1 + sp_2)(A) = \{B \in \text{Alg}(\Sigma_{1e} \cup \Sigma_{2e}) \mid (B|_{\Sigma_{1e}}) \in \mathcal{M}(sp_1)(A|_{\Sigma_{1i}}) \wedge (B|_{\Sigma_{2e}}) \in \mathcal{M}(sp_2)(A|_{\Sigma_{2i}})\};$$
  - If  $\mathcal{S}(sp_1) = (\Sigma_i, \Sigma)$  and  $\mathcal{S}(sp_2) = (\Sigma, \Sigma_e)$ , then
$$\mathcal{M}(sp_2 \circ sp_1)(A) = \bigcup_{B \in \mathcal{M}(sp_1)(A)} \mathcal{M}(sp_2)(B);$$
  - If  $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$ , then
$$\mathcal{M}(\text{rename } sp \text{ by } \mu)(A) = \{B \in \text{Alg}(\mu(\Sigma_e)) \mid (B|\mu) \in \mathcal{M}(sp)(A)\};$$
  - The semantics of the constructs **forget**, **extend**, **model**, and **(freely) generated** is defined similarly as before.

Generalization of the semantics of a specification from an ADT to a function that takes an algebra and returns an ADT.

# Example

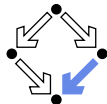


As shown in previous section, also module specifications may be named.

```
BOOL is
  loose mspec
    sorts freely generated bool
    opns constr True  $\rightarrow$  bool, False  $\rightarrow$  bool
  endmspec;
EL is loose mspec sorts el endmspec;
LIST is ...; (see last example)
LIST  $\circ$  (BOOL + EL)
```

Since the import signature of this specification is empty, it may be considered as a specification with signature  $(\{bool, el, list\}, \{True, False, [], Add\})$ .

# Properties

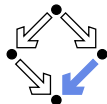


Take specification  $sp \in MSP$  with  $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$ .

- $\mathcal{M}(sp)$  maps  $\Sigma_i$ -algebras to classes of  $\Sigma_e$ -algebras.
- $\mathcal{M}(sp)(A)$  is an abstract datatype, for each  $\Sigma_i$ -algebra  $A$ .
- Each construct of the module specification language preserves persistency.
  - Thus any module specification is persistent, provided that the atomic specifications in it are.
- Each construct of the module specification language except **model**, **generated**, and **freely generated** preserves consistency.
  - Thus any module specification that does not use these constructs is consistent, provided that the atomic specifications in it are.

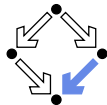
The semantics of the module specification language is “as expected”.

# Import Signatures Revisited



What is actually the purpose of a specification's import signature?

- Consider  $LIST \circ (BOOL + \dots)$ 
  - $LIST$  uses an imported sort  $bool$ .
  - $BOOL$  provides a specification of this sort.
  - Purpose: we want to reuse  $bool$  in different contexts.
    - Only a single specification  $BOOL$  suffices; its can then be used by import in multiple other specifications.
- Consider  $LIST \circ (\dots + EL)$ 
  - $LIST$  uses an imported sort  $el$ .
  - But we actually do not expect a specification for  $el$  !
  - Rather  $el$  saves as a “placeholder” for some *other* sort.
  - Purpose: we want to instantiate  $el$  by different sorts.
    - Only a single specification  $LIST$  suffices; its sort  $el$  can then be instantiated by multiple concrete sorts.
  - Two additional mechanisms are needed:
    - A mapping of the specified sorts to the actual sorts.
    - A mean to express semantic constraints on the imported sorts.



---

## 1. A Specification Language

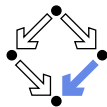
## 2. Modularization

## 3. Parameterization

## 4. Further Topics



# Parameterized Specifications



We extend module specifications to parameterized specifications.

- **Abstract Syntax:** set  $PSL$  of specifications  $sp$  with signatures  $\mathcal{S}(sp)$ .
    - If  $sp \in PSL$  with  $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$  and if  $\mu : \Sigma_i \cup \Sigma_e \rightarrow \Sigma'$  is a signature morphism that “renames the import signature”, i.e.
      - $\mu(s) = s$  for each sort  $s \in \Sigma_e \setminus \Sigma_i$ ,
      - $\mu(\omega)$  and  $\omega$  have the same operation name for each op.  $\omega \in \Sigma_e \setminus \Sigma_i$ , and that avoids “name clashes” with introduced sorts, i.e.
      - $\mu(a) = \mu(b)$  implies  $a$  and  $b$  are inherited, for all  $a, b \in \Sigma_e, a \neq b$ ,
      - $\mu(a) = \mu(b)$  implies  $b$  is inherited for each  $a$  from  $\Sigma_i$  and  $b$  from  $\Sigma_e$ ,
- then

**(import rename  $sp$  by  $\mu$ )**  $\in PSP$

with  $\mathcal{S}(\mathbf{import\ rename\ } sp \mathbf{\ by\ } \mu) = (\mu(\Sigma_i), \mu(\Sigma_e))$ ;

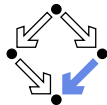
- If  $sp \in PSP$  with  $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$  and  $\Phi \subseteq L(\Sigma_i)$  for logic  $L$ , then

**( $sp$  import model  $\Phi$ )**  $\in PSP$

with  $\mathcal{S}(sp \mathbf{ import\ model\ } \Phi) = \mathcal{S}(sp)$ ;

- ... (as before using  $PSL$  rather than  $MSL$ ).

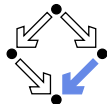
# Example



Take  $\Sigma_i = (\{a, b\}, \emptyset)$ ,  $\Sigma_e(\{a, c\}, \emptyset)$ .

- A signature morphism  $\mu$  suitable for **import rename** must *not* allow
  - $\mu(c) = d$ ,
    - First condition is violated.
    - $\mu$  renames an entity introduced by the specification.
  - $\mu(a) = \mu(c)$ ,
    - Third condition is violated.
    - $\mu$  maps exported sort  $a$  to the same name as the introduced sort  $c$ .
  - $\mu(b) = \mu(c)$ .
    - Fourth condition is violated.
    - $\mu$  maps imported sort  $b$  to the same name as the introduced sort  $c$ .

The signature morphism is intended to map actual “argument” sorts to formal “parameter” sorts.



- **Semantics:**  $\mathcal{M}(sp)$  is inductively defined:

- If  $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$ , then for each  $A \in \text{Alg}(\mu(\Sigma_i))$

$$\mathcal{M}(\text{import rename } sp \text{ by } \mu)(A) = \{B \in \text{Alg}(\mu(\Sigma_e)) \mid (B|_{\mu|_{\Sigma_e}}) \in \mathcal{M}(sp)(A|_{\mu|_{\Sigma_i}})\};$$

- Let  $f : A \rightarrow B$  and  $C \subseteq A$ . The **restriction**  $f|_C$  is the function

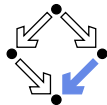
$$\begin{aligned} f|_C &: C \rightarrow B \\ f|_C(c) &= f(c) \end{aligned}$$

- If  $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$ , then for each  $A \in \text{Alg}(\mu(\Sigma_i))$

$$\mathcal{M}(sp \text{ import model } \Phi)(A) = \begin{cases} \mathcal{M}(sp)(A) & \text{if } A \models \Phi \\ \emptyset & \text{otherwise} \end{cases};$$

- ... (as with module specifications).

# Properties

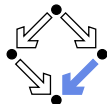


Take specification  $sp \in PSL$  with  $\mathcal{S}(sp) = (\Sigma_i, \Sigma_e)$ .

- $\mathcal{M}(sp)$  maps  $\Sigma_i$ -algebras to classes of  $\Sigma_e$ -algebras.
- $\mathcal{M}(sp)(A)$  is an abstract datatype, for each  $\Sigma_i$ -algebra  $A$ .
- **import rename** and **import model** preserve persistency.
- Only **import rename** preserves consistency.

The semantics of the parameterized specification language is “as expected”.

# Example



## Parameterized specification

**loose pspec**

**sorts** **import**  $el_1$ , **import**  $el_2$ , **freely generated**  $pair$

**opns**

**constr**  $[-, -] : el_1 \times el_2 \rightarrow pair$

$First : pair \rightarrow el_1$

$Second : pair \rightarrow el_2$

**vars**  $e_1 : el_1, e_2 : el_2$

**axioms**

$First([e_1, e_2]) = e_1$

$Second([e_1, e_2]) = e_2$

**endpspec**

defines a  $(\Sigma_i, \Sigma_e)$ -module with

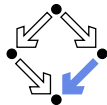
$\Sigma_i = (\{el_1, el_2\}, \emptyset)$ ,

$\Sigma_e = (\{el_1, el_2, pair\}$ ,

$\{[-, -] : el_1 \times el_2 \rightarrow pair, First : pair \rightarrow el_1, Second : pair \rightarrow el_2\})$ .

**Specification of  $(el_1, el_2)$ -pairs.**

## Example (Contd)



Parameterized specification

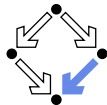
```
PAIR is loose pspec ... endpspec;  
import rename PAIR by sorts el1, el2 as sorts nat, nat
```

defines a  $(\Sigma_i, \Sigma_e)$ -module with

```
 $\Sigma_i = (\{\text{nat}\}, \emptyset),$   
 $\Sigma_e = (\{\text{nat}, \text{pair}\},$   
   $\{[-, -] : \text{nat} \times \text{nat} \rightarrow \text{pair}, \text{First} : \text{pair} \rightarrow \text{nat}, \text{Second} : \text{pair} \rightarrow \text{nat}\}).$ 
```

Specification of *nat*-pairs.

## Example (Contd'2)



Parameterized specification

```
PAIR is loose pspec ... endpspec;
```

```
NAT is loose pspec
```

```
  sorts freely generated nat
```

```
  opns
```

```
    constr 0 :→ nat
```

```
    constr Succ : nat → nat
```

```
  endspec;
```

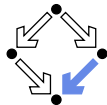
```
(import rename PAIR by sorts el1, el2 as sorts nat, nat) ◦ NAT
```

defines a module with empty import signature and export signature

$$\Sigma = \{nat, pair\},$$
$$\{[-, -] : nat \times nat \rightarrow pair, First : pair \rightarrow nat, Second : pair \rightarrow nat\}.$$

Specification of pairs of natural numbers.

## Example (Contd'3)



Better notation for parameterized specifications:

*PAIR*(**sorts**  $el_1, el_2$ ) **is loose pspec ... endpspec**;

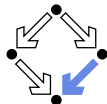
*NAT* **is loose pspec ... endpspec**;

*PAIR*(**sorts**  $nat, nat$ )  $\circ$  *NAT*

Similar to definition and application of parameterized procedures.

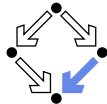


# Example



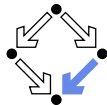
*OLISTS*(*sorts* *el*, *opns*  $\_ \sqsubseteq \_ : el \times el \rightarrow bool$ ) is  
(*loose pspec*  
  *sorts* *import* *bool*, *import* *el*, *freely generated list*  
  *opns*  
    *import* *True*  $: \rightarrow bool$   
    *import* *False*  $: \rightarrow bool$   
    *import*  $\_ \sqsubseteq \_ : el \times el \rightarrow bool$   
    *constr*  $[ ] : \rightarrow list$   
    *constr* *Add*  $: el \times list \rightarrow list$   
  *vars* *e*, *e*<sub>1</sub>, *e*<sub>2</sub>  $: el$ , *l*  $: list$   
  *axioms*  
    *ordered*( $[ ]$ ) = *True*  
    *ordered*(*Add*(*e*,  $[ ]$ )) = *True*  
    (*e*<sub>1</sub>  $\sqsubseteq$  *e*<sub>2</sub>) = *True*  $\Rightarrow$  *ordered*(*Add*(*e*<sub>1</sub>, *Add*(*e*<sub>2</sub>, *l*))) = *ordered*(*Add*(*e*<sub>2</sub>, *l*))  
    (*e*<sub>1</sub>  $\sqsubseteq$  *e*<sub>2</sub>) = *False*  $\Rightarrow$  *ordered*(*Add*(*e*<sub>1</sub>, *Add*(*e*<sub>2</sub>, *l*))) = *False*  
  *enspec*)  
  *import model*  
    *vars* *e*, *e*<sub>1</sub>, *e*<sub>2</sub>, *e*<sub>3</sub>  $: el$   
    *axioms*  
      (*e*  $\sqsubseteq$  *e*) = *True*  
      (*e*<sub>1</sub>  $\sqsubseteq$  *e*<sub>2</sub>) = *True*  $\wedge$  (*e*<sub>2</sub>  $\sqsubseteq$  *e*<sub>3</sub>) = *True*  $\Rightarrow$  (*e*<sub>1</sub>  $\sqsubseteq$  *e*<sub>3</sub>) = *True*  
      (*e*<sub>1</sub>  $\sqsubseteq$  *e*<sub>2</sub>) = *True*  $\wedge$  (*e*<sub>2</sub>  $\sqsubseteq$  *e*<sub>1</sub>)  $\Rightarrow$  *e*<sub>1</sub> = *e*<sub>2</sub>

## Example (Contd)



```
OLISTS(sorts el, opns  $\_ \sqsubseteq \_ : el \times el \rightarrow bool$ ) is
...;
NATBOOL is
  loose pspec
    sorts freely generated bool, freely generated nat
    opns
      constr True  $: \rightarrow bool$ 
      constr False  $: \rightarrow bool$ 
      constr 0  $: \rightarrow nat$ 
      constr Succ  $: nat \rightarrow nat$ 
       $\_ \leq \_ : nat \times nat \rightarrow bool$ 
    vars m, n  $: nat$ 
    axioms
       $(0 \leq n) = True$ 
       $(Succ(m) \leq 0) = False$ 
       $(Succ(m) \leq Succ(n)) = (m \leq n)$ 
    endpspec;
  OLISTS(sorts nat, opns  $\leq : nat \times nat \rightarrow bool$ )  $\circ$  NATBOOL
```

Specification of ordered list of natural numbers; specification is adequate, because  $\leq$  satisfies the axioms imposed on  $\sqsubseteq$



---

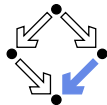
## 1. A Specification Language

## 2. Modularization

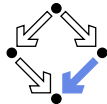
## 3. Parameterization

## 4. Further Topics

# Open Issues



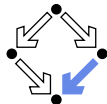
- Constructs **extend** and **model** have loose semantics.
  - Initial semantics counterparts require the notion of “free extensions” .
    - Generalization of the notion of “initial algebra” .
    - Algebras in free extension have common “stem” which does not “take part” in initiality.
  - Initial counterpart of **extend** is (**freely extend**  $sp$  **by**  $(S, \Omega)$ ).
    - Constructs only free extensions (rather than all extensions).
  - Initial counterpart of **model** is ( $sp$  **quotient**  $\Phi$ ).
    - Builds quotient algebras (rather than removing algebras).
- Specifications can be flattened.
  - Compound specifications can be translated to equivalent atomic ones.
- There exist alternative parameterization mechanisms.
  - We have used the *renaming approach* with a syntactic flavor.
  - There exist approaches with a semantic flavor.
    - Based on  $\lambda$ -calculus or on category theory.
  - However, all approaches are ultimately equivalent in expressive power.



CafeOBJ supports some of the described constructions.

- Named modules:
  - ***n* is loose (initial) spec ... endspec**  
module\* (module!) *n* { ... }
  - ***n* is ...** (arbitrary module expression)  
make *n* (...)
- References to named modules: *n*  
*n*
- Union:  $sp_1 + sp_2$   
SP1 + SP2
- Renaming: **rename *sp* by ...**  
SP \* { sort *s1* -> *s1'* op *w1* -> *w1'* ... }
- Extension and Modelling: ***sp* extend ... model ...**  
protecting (SP) signature { ... } axioms { ... }
- ...

# CafeOBJ (Contd)



- ...
- Parameterized Modules
  - Parameters are whole modules (rather than sorts or operations).

```
module* SP1 { [ s1 ... ] op o1: ... }  
module* (module!) SP (P1::SP1, ...) { ... }
```
- Module Instantiation
  - “Views” specify bindings of actual arguments to formal parameters.

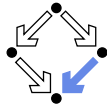
```
module! SP2 { [ s2 ... ] op o2: ... }  
view V from SP1 to SP2 { sort s1 -> s2, op o1 -> o2, ... }
```
  - Instantiation of parameter module by a declared view

```
SP(P1 <= V1, ...)
```
  - Instantiation of parameter module by ad-hoc view

```
SP(P1 <= view to SP2  
  { sort s1 -> s2, op o1 -> o2. ... }, ...)
```

See the CafeOBJ manual for more details

# Parameterized Modules in Programming



Parameterized modules are now part of various programming languages.

- **ML functors**

```
signature ELEM = sig ... end;  
functor STACK(structure EL: ELEM) = struct ... end;
```

- **C++ templates** (type checking only after instantiation)

```
template <class EL> class Stack { ... }
```

- **Java generic types**

```
interface ELEM { ... }  
class Stack<EL implements ELEM> { ... }
```

- **C# generic types**

```
interface ELEM { ... }  
class Stack<EL> where EL:ELEM { ... }
```

- ...